14P

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence
Memo No. 34

A New Eval Function

John McCarthy

This empty page was substituted for a blank page in the original document.

The actual working definition of eval describes how the LISP system determines what, if anything, is denoted by a given S-expression. As things now stand, there are two versions of eval: the theoretical version, given in RFSE, and the system version. Neither of these behaves in the most desirable way; and there exist S-expressions which will be handled correctly by the theoretical version but not by the system version, and conversely. The chief defect of the system eval lies in its handling of functional arguments; the chief defect of the RFSE eval lies in its ignorance of property lists. If we wish to have a theory about how LISP really works, then it is necessary to have a version of eval which is satisfactory both theoretically and practically. I will propose a definition for eval, and then illustrate how this eval differs from the existing system and RFSE definitions by means of examples.

Consider the following definition:

```
eval (exp; alist) = (..atom(exp) → search (.exp;
        λ((j);(eq(car(j); VALUE)));
          cadr;
            λ((); assoc(exp; alist.)
t → prog((fnval);
    fnval = eval (car(exp); alist)
```

```
    return((fullword(fnval) ∨ eq(car(fnval);LAMBDA)
      ∨eq(car(fnval);LABEL) → app 1 (fnval;
    maplist(,cdr(exp); λ((j);eval(car(j); alist,); alist);
      t → app 1 (car(fnval); list(cdr(exp);alist);alist..)
app 1(fn;args;alist) = (.
      fullword (fn)→ app 2 (fn; args);
        eq(car(fn);LAMBDA)→ eval(caddr(fn);append(
        pair(cadr(fn);args);alist));
         eq (car(fn);LABEL) →app 1 (caddr(fn); args;
        cons (cons(cadr(fn);caddr(fn)); alist.)
evalquote(fn; args) = app 1 (eval (fn;nil)args;nil)
```

There are some new notational conventions used in this definition. Brackets may be marked by an arbitrary sequence of dots and commas. The mark for a left bracket follows the bracket; the mark for a right bracket closes out the rightmost enclosed left bracket with the same mark to the left of the right bracket and all intermediate brackets. This rule works even for the case of unmarked brackets, which are assumed to have an empty mark. Thus, the last right bracket on the fourth line of eval is marked with a period. Moving to the left, we find that the first left bracket we encounter which is unclosed and marked with a period is the one after search on the first line. (If we had encountered other left

brackets marked with a period in between, we would ignore them if they had already been closed out.) This right bracket closes the left bracket after search, the left bracket after the λ on the fourth line, and the left bracket after assoc on that line. The right bracket at the end of the last line of eval closes out the entire function definition.

Two other changes of notation are the use of t instead of T to denote truth, and the use of cadr without enclosing it in a λ on the third line of eval. The use of t is a consequence of the rules for going from M-expressions to S-expressions. If we used T, the proper translation would be (QUOTE, T); but t is properly translated as T, which is what we want. A similar situation holds for nil and NIL. The use of cadr rather than λ ((j;cadr(j)) is certainly more convenient, and is justified if eval is evaluated according to its own definition, using the usual rules for going from M-expressions to S-expressions.

Although eval is intended to operate on more or less the same set of S-expressions as the system eval and the RFSE eval, the function appl differs from apply. The definition of evalquote as given in terms of appl shows how appl relates to current conventions.

In order to make eval work correctly, certain assumptions must be made about the underlying structure. In the actual system, this means that the right things must be on property lists. In the

theoretical description, we may either axiomatize those characteristics of property lists which we need, or specify a permanent a-list which permits atoms to remain unanalysable. If we adapt the second course, we also define the predicate fullword to be always false, and define search so as to evaluate its fourth argument (the escape case) whenever the first argument (the list to be searched) is atomic. An axiomatization of LISP with property lists is given in an appendix.

The functions assoc, search, among, pair, maplist, and append have their usual definitions, and the a-list has the usual form. The function app 2 applies a machine-language function to its arguments. The indicator value precedes the value of an atom on its property list; the value may be an S-expression or may involve a pointer to a TXL word. The predicate fullword can be used to detect the second situation; fullword ($\mathfrak{a}$) is true if $\mathfrak{a}$ points to a full word, and false otherwise.

The value of a function will be one of two types, depending on whether the arguments of the function are to be evaluated first or not. If, as is the case with most ordinary functions, the arguments are to be evaluated first, the value will be either a pointer to a TXL word or a list beginning with LABEL or a list beginning with LAMBDA. If the arguments of the function must be given without evaluation, then the value is of the form (FEXPR.$\mathfrak{d}$), where $\mathfrak{d}$

is either a pointer to a TXL word or a list beginning with LABEL or a list beginning with LAMBDA.

In this version of eval unlike the RFSE version the elementary functions are not explicitly built into the definition. Function definitions are now all on the property lists, though for theoretical purposes we may simulate the effect by associating the elementary function names with their definitions on a permanent a-list. The permanent a-list replaces nil in the cal for appl in the definition of evalquote. eval does not need to be changed when function definitions are transferred between the permanent a-list and property lists. To illustrate, the word following VALUE on the property list of COND points to the S-expression

((LAMBDA (C A) (EVCON C A)) .FEXPR)

and the word following VALUE on the property list of EVCON points to the S-expression

((LAMBDA (C A) (COND ((EVAL (CAAR C) A) (EVAL (CADAR C) A))
(T (EVCON (CDR C) A)))))

or alternatively the value of COND might be

((LABEL EVCON (LAMBDA (C A) ...)).

We could just as well append these to the permanent a-list, of course. Or, to be practical, we would make the value of COND (α. FEXPR), where α points to a TXL to a machine language COND program. In order to account for the case where a S-expression

is explicitly given as a function, we must make LAMBDA into an autonym (and LABEL also). (An autonym is an expression which is its own name, i.e. its value is itself.) This adds new flexibility to the language, and brings several inconsistencies into line with the theory without introducing other theoretical difficulties. To declare LAMBDA an autonym, we make its value

((LAMBDA (E A) (CONS (QUOTE LAMBDA) E)).FEXPR).

Thus, any S-expression beginning with (LAMBDA (but not with ((LAMBDA!) will evaluate to itself.

SUPPOSE that we have an S-expression of the form (fn arg 1 arg 2). The RFSE eval and the new eval permit fn to be either a LAMBDA expression (for the sake of simplicity we will not consider LABEL, which works much like LAMBDA) or an atom which denotes a LAMBDA expression. The system eval permits both of these cases, and in addition permits the case where fn is an expression which evaluates to a LAMBDA expression or to an atom denoting a LAMBDA expression. However, neither the RFSE eval nor the system eval makes provision for LAMBDA expressions being autonyms in general, and this causes difficulty when LAMBDA expressions are used as functional arguments. Neither the system eval nor the RFSE eval gives a consistent answer to the question, "What is denoted by fn in the above example?" The RFSE eval says that the denotation

depends on whether fn is atomic or whether it is a LAMBDA expression; if it is atomic, it denotes what it is paired with on the a-list (just as if it had occurred in an argument position), but if it is a LAMBDA expression, it denotes itself (unlike what happens in argument position). The system eval is even worse: the value indicators SUBR, FSUBR, etc. are considered in the functional position to give the denotation of fn, but will not be examined in an argument position; and conversely for the indicator APVAL: if fn has both an APVAL and an EXPR pointing to different definitions on its property list, then the EXPR definition will be taken in the function position and the APVAL definition in the argument position; but if the EXPR definition is removed, the APVAL definition will be taken in both instances! The new eval, on the other hand, permits only one type of value indication, and this will be detected in any context. By placing appropriate things on property lists most programs written in existing notation can be made to work as intended.

This situation with the system and RFSE evals might not be so unpleasant, and in fact would even have advantages, were it not for functional arguments. It is often convenient to let the same name denote more than one thing -- this is done in mathematics all the time. But this can be done safely only when it is clear from context what the name denotes. If the atom a has both an EXPR and an

APVAL on its property list, then we must decide what to do when *a* appears as an argument to be evaluated. If it is functional, we really want the EXPR: if it is not, we want the APVAL. But we cannot just tell from context whether *a* is a functional argument or not. Thus the system eval will accept functional arguments only when they are preceded by FUNCTION, which is both a theoretical and practical nuisance.

To illustrate in terms of a specific example, suppose that x is bound, via the a-list or a property list, to ((A C)(B D)), and we wish to evaluate

maplist [x;caar] .

If we translate this M-expression into an S-expression, we get

(MAPLIST X CAAR).

If we apply the new eval to this, we get (A B), which is what we want. If we apply the system eval, CAAR looks like an unbound variable and we get an error complaint. If we apply the RFSE eval, we get the right thing provided that caar is bound to its definition on the a-list; however, even if we replaced caar by car, we would need to bind car to

(LAMBDA (X) (CAR X))

on the a-list; i.e.

(MAPLIST X CAR)

would be undefined as far as the RFSE eval is concerned. The

difficulties are not simply due to the use of caar standing by itself, however, for suppose we try to evaluate

maplist [x;λ [[j] caar [j]]].

Translating into an S-expression, we get

(MAPLIST X (LAMBDA (J) (CAAR J))).

Again, the new eval gives the right result with no difficulty; and both the system eval and the RFSE eval complain that LAMBDA is an undefined function! The RFSE eval will, however, accept

(MAPLIST X (QUOTE (LAMBDA (J) (CAAR J)))),

and the system eval will accept

(MAPLIST X (FUNCTION (LAMBDA (J) (CAAR J)))).

Neither of these are as convenient as the simple version; but the new eval will accept them both anyway (again provided that the appropriate thing is put on the property list of FUNCTION).

The difficulties with the existing versions of eval have turned out to be somewhat of an obstacle in the Proofchecker. In the Proofchecker, I have on hand an S-expression which denotes what it is sufficient to prove. Subexpressions of this expression may denote other S-expressions or may denote expressions which denote S-expressions. It is important to be able to distinguish linguistic levels, and to operate on expressions which may be one or more levels of denotation below the one at hand. In other words, if A denotes B and B denotes C, then we may need to infer properties of

C from A. In order to make this kind of inference, it is necessary to have a more tractable eval than the one in the system. At the same time, it makes the task much easier to be able to use property lists and machine-language subroutines, and there is no provision for these in the RFSE eval. I hope to be able to get around these difficulties by the use of the new eval.

APPENDIX: AXIOMATIZATION OF EXTENDED LISP

Extended LISP is an attempt to describe with some rigor the behavior of those parts of the LISP operating system which concern the Proofchecker, namely machine-language subroutines and property lists. For this description, we take the original mathematical description of LISP and append three new elementary functions and a new class of expressions. We define two of the new elementary functions by enumeration, i.e. we define the function by stating explicitly for each possible argument the value of the function. We also augment the definitions of the elementary functions and predicates to account for the new class of expressions. The actual details depend on the current state of the system, but the form is independent of these details. We will not include an axiomatization of the integers as they appear in LISP; however, this could probably be done within the same framework.

First, we define the class of full word pointers by enumeration. These consist of all pointers to full words on property lists in the system version being considered. We denote a pointer to the full word a [as an octal number) | by: d, e.g. *300000235572. We then define the class of extended symbolic expressions (E-expressions) as follows:

1. An atomic symbol is an E-expression.

2. A full word pointer is an E-expression.

3. If e_1 and e_2 are E-expressions, then $(e_1 \cdot e_2)$ is an E-expression.

The elementary functions car and cdr are undefined for full word pointers and are otherwise defined as usual. The elementary function cons may have a full word pointer as an argument. The predicate atom is false for full word pointers and defined otherwise as usual. The predicate eq is true for two full word pointers to the same octal word (which are, of course, the same sequence of characters). We also introduce the new elementary predicate fullword[x] which is defined for all E-expressions and is true when x is a full word pointer, false otherwise.

We next define the function value [x] by enumeration. value[x] is defined only when x is an atomic symbol. If x has the value v on its property list, then value[x] = (VALUE.v); otherwise NIL. We assume that the system does not vary at running time. The value v is an E-expression.

Finally, we define the function app2[fn;args] where args is an S-expression and fn is a full word pointer to a subroutine. Again, we simply enumerate all machine-language subroutines and for each one give a description of its effect and its domain of definition, possibly by an equivalent λ-definition.

These additions to LISP provide all that is needed for the new eval to work. The only change needed is that search should be replaced by a function which does not explicitly search a property list, but rather uses the function value.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 11/30/95

Report # AIM-34

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 16 (20 - IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☒ Single-sided or
☐ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-16) UNNUM'D TITLE & BLANK PAGES, 1-13, UNNUM'D
(17-20) SCANCONTROL, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11/30/95 Date Scanned: 12/12/95 Date Returned: 12/14/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/12/1995

M.I.T. Libraries
Document Services